SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 8, 229–239 (1996)

Research

The GUMP Process for Software Maintenance and Maintenance Education

NORMAN WILDE
Department of Computer Science, University of West Florida, Pensacola, FL 32514, U.S.A.

SCOTT M. BROWN
Department of Electrical and Computer Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH, 45315, U.S.A.

SUMMARY

Software producing organizations attempt to improve their cost, schedule and quality performance by defining an organizational software process model. Educators try to teach their students the virtues of following such a process in software development and maintenance. But most software process models are proprietary so there are few readily available examples to help guide these efforts.

This paper outlines the 'Generic University of West Florida Maintenance Process (GUMP)' which has been developed, used and refined over a period of almost two years in an educational setting. It describes the key maintenance process issues identified in this experience and the rationale for the solutions adopted in GUMP. As an academic tool, GUMP has greatly improved the learning experience in our project courses; two of our graduates have taken it as a starting point for developing a process for their employers.

Full documentation for GUMP is publicly available on-line and may be copied and adapted as needed. GUMP is presented not as a 'perfect' process, but simply as a starting point that may be useful to either maintainers or educators as a point of reference for software process concepts.

KEY WORDS: software maintenance; software process model; process architecture; standards; management

INTRODUCTION

It is now generally accepted that the first requisite for improving the timeliness and quality of an organization's software products is a mature software process. The establishment of a defined software process is, in some markets, becoming a requirement of doing business.

The new emphasis on process creates challenges for both software engineering educators and for software development organizations, and unfortunately there is still little published information to guide them. 'Process' is a very difficult subject to discuss in the abstract. Educators teaching about process need to have a range of processes to which they can point. Managers establishing a process in their organizations likewise need example processes as starting or reference points.

But examples of successful, fully elaborated processes are hard to come by. Most

CCC 1040–550X/96/040229–11

Received 21 September 1995
Revised 29 April 1996

companies that have expended the effort to define and validate their own software process regard the results as proprietary, and well elaborated processes for use in the classroom are still very scarce (though see Humphrey (1995) for an excellent step in this direction).

This paper describes GUMP, the 'Generic UWF Maintenance Process' that has been developed, used, and refined over the last 20 months by University of West Florida students in the Masters in Computer Science—Software Engineering option at the Fort Walton Beach campus. This paper itself provides a high-level description of the process, as well as its history and context of applicability. Complete GUMP documents, amounting to approximately 150 pages, are available electronically as indicated at the end of the paper.

We present GUMP not as a 'perfect' process, but simply as an example, freely available and at least moderately successful, that maintenance organizations and maintenance educators may use as a point of reference or a point of departure for their own process definition efforts. At least two of our graduates have started to use it as a basis for defining processes in their employer's organizations.

THE GUMP MAINTENANCE PROCESS

The software engineering program at the University of West Florida is taught in the evenings; most of the students at the Fort Walton campus have full time jobs related to software activities at a nearby Air Force Base. A sequence of three semester-long courses is orientated towards software process. The first, *Software Engineering Management*, focuses on the Software Engineering Institute's Capability Maturity Model (Paulk, *et al.*, 1993), as well as on the practicalities of configuration management, quality assurance and project management. Then students take two semesters of *Software Engineering Project*, in which a single team of approximately 12 students works together to develop or maintain a small software system.

The GUMP software maintenance process was first developed by students in the Summer, 1994 session of *Software Engineering Management*. The process was then put to use in the 1994/95 and 1995/96 *Software Engineering Project* classes. In each case, the process was refined to incorporate the practical experiences gained.

The GUMP process has chiefly been used in enhancing the software tool RECON2, a dynamic tool to aid maintainers in understanding unfamiliar code (Wilde and Scully, 1995). The process is thus designed for making a series of fairly small progressive changes to a modest-sized system written in the C language and running under MS-DOS and Unix. Other environments or objectives would certainly require some modifications to GUMP.

It should be emphasized that many of the students who developed and refined GUMP are experienced professionals who have encountered, at least tangentially, the United States Department of Defense's efforts to improve software quality through standardization and process maturity. In their jobs, some students were software developers, others were testers, others had quality assurance or management functions. The sharing of experiences and points of view made process development an exciting and sometimes rather exuberant activity, and a most unusual academic exercise!

The team organization used in GUMP is shown in Figure 1. The course instructor serves as 'client' to channel requests for software change and to help set priorities. The Project Co-ordinator handles day-to-day management of the team's activities and runs the weekly meetings. Remaining team members are assigned roles as:

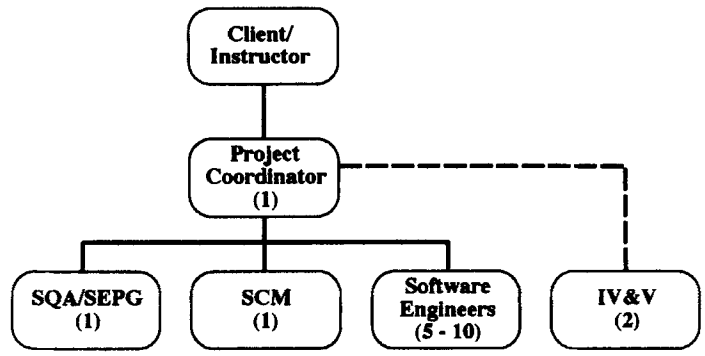

Figure 1. Organization used during the project

SQA/SEPG—Software Quality Assurance and Software Engineering Process Group
SCM—Software Configuration Management
Software Engineers—Responsible for coding and unit testing
IV&V—Independent Verification and Validation

The overall structure of the GUMP process architecture is shown in Figure 2. The process is driven by *Change Requests* submitted by either customers or team members suggesting improvements or bug fixes in the software system. *Change Requests* are given a brief sanity check and, if they pass, filed as *Deficiencies* (cell 100). When resources are available, the Project Co-ordinator selects one or more related *Deficiencies* for analysis, thus initiating a *tisk*, as a complete cycle of software change is called (cell 200).[1]

A *tisk* starts with an analysis step (cell 300) which defines requirements and high-level design for the software change, along with a risk analysis and an estimate of the resources needed for implementation and testing. The final product of this cell is an *Analysis Report* which is subjected to an inspection supervised by the Software Quality Assurance team.

The approved *Analysis Report* then goes to the Change Control Board (CCB), composed of the Client, the Project Co-ordinator, and one member each from Independent Verification and Validation, Software Quality Assurance, Software Configuration Management and Software Engineering (cell 400). Here the basic decision to commit resources to the change is made. If the decision is positive, Software Configuration Management allows the Software Engineers to check out any code and needed documents from the configuration management system (cell 500).

Change implementation (cell 600) consists of making the changes to code and documents, unit testing, and a final inspection again supervised by Software Quality Assurance. The work then goes to Independent Verification and Validation, which consists mainly of integration and system level testing (cell 700). If testing is successful, the revised software is checked back in to the configuration management system (cell 800).

Most of the high-level process cells shown in Figure 2 have been further broken down

[1] The cycle was initially called a 'task', but was then frequently confused with the tasks in each process cell. The student group adopted the coined name *tisk*, and simultaneously took as the group's symbol a green and yellow basket.

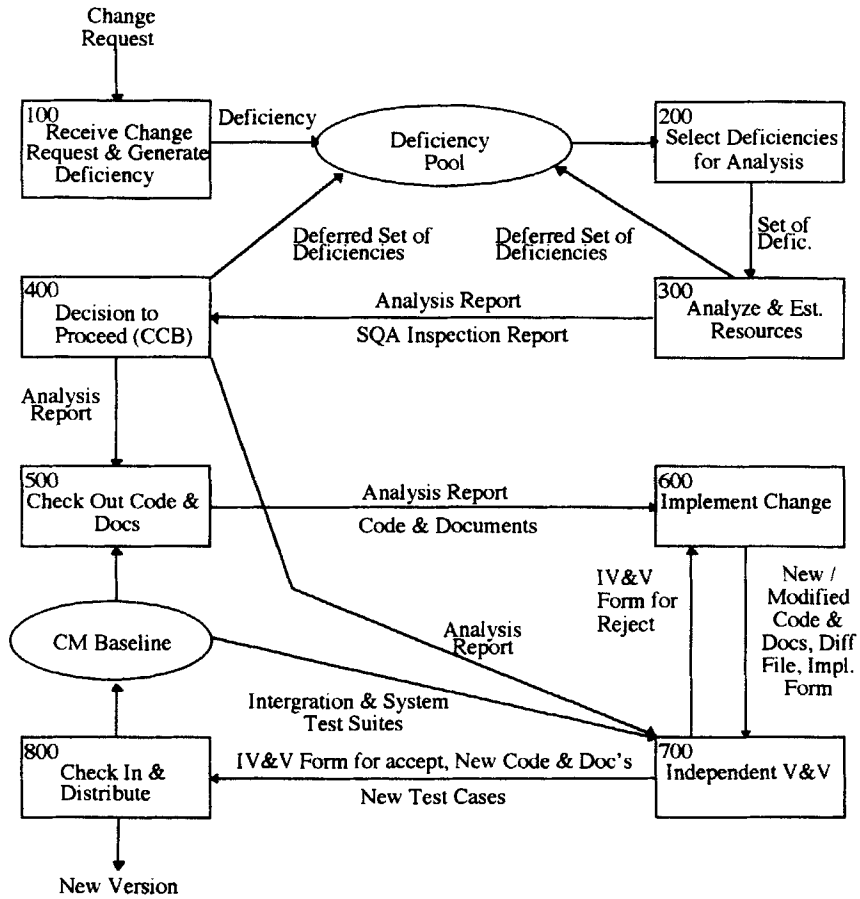

Figure 2. Overall structure of the maintenance process

into more detailed cells. For example, Figure 3 shows the breakdown of cell 600 (*Implement Change*).

At the lowest level, each cell is described using an 'Entry–Task–Exit (ETX)' format adapted from that suggested by Watts Humphrey (Humphrey, 1989). As an example, Table 1 shows the description of cell 601, '*Unit Testing*'. The tasks for the cell are briefly listed and the group responsible (Software Engineers, SQA, etc.) is identified. Cell task descriptions may reference GUMP standards documents that provide detailed guidance on how a task is to be carried out. The unit testing standard, for example, states that the 'Automatic Test Analysis for C (ATAC)' coverage tool (Horgan, London and Lyu, 1994) should be used to check that every testable block and decision is covered, and that non-testable decisions, such as tests for operating system errors, should be explicitly hand checked. GUMP also includes standard forms to be used in most of the main tasks.

Additionally, certain cells have associated metrics that are collected, along with weekly timecard data, and kept in a central database, available to all team members. The data are intended to be used to make better and more accurate time estimations and schedules for *tisk* milestones. Finally, some cells have required training that must be completed in

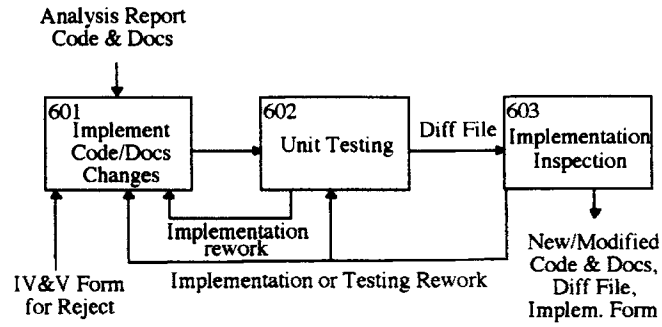

Figure 3. Breakdown of cell 600—'Implement Change'

Table 1. Entry–task–exit description of the 'unit test' cell

Cell	602 – Unit test
Entry (from cell)	–(601) New/modified code and docs
Exit (to cell and group)	–(603) New/modified code and docs –(603) New tests (as required) –(603) Diff file –(603) Implementation form
Feedback in	–(603) Testing rework
Feedback out	–(601) Implementation rework
Tasks (responsible group)	– Conduct unit testing IAW testing std. (SE) – Generate diff file and implementation fm. (SE) – Attach diff file and detailed design to implementation fm. (SE) – Inform PC ready for inspection (SE)
Measures	
Training required	– Project orientation, ATAC

order to accomplish the subcell tasks. Training is conducted during the first weeks of the semester in accordance with a training plan.

ISSUES IN THE DESIGN OF SOFTWARE MAINTENANCE PROCESSES

Most of the published material on software process has focused on development, not maintenance. However a few recent reports have discussed maintenance processes. Drew has described how the Capability Maturity Model may be tailored for maintenance (Drew, 1992), while Hinley and Bennett provide a framework for process development (Hinley and Bennett, 1992). Several of these papers have concentrated on a particular software maintenance technique or tool, and present a process to provide context showing how the technique or tool is intended to be applied. For example, Cherinka and others describe a process-orientated approach to developing a software engineering environment (Cherinka,

et al., 1994) while Capretz and Munro present a process as a framework for defining an incremental documentation and configuration management approach to maintenance (Capretz and Munro, 1994).

Perhaps the most detailed process guidance that has been published comes from the EPSOM project and from the IEEE. As part of the European Platform for SOftware Maintenance (EPSOM), Harjani and Queille present what they call a 'generic' maintenance process. This paper defines key terms needed in describing software maintenance activities, and describes the broad steps in maintenance and the participants in each step (Harjani and Queille, 1992). The steps are: trigger, problem understanding, localization, solution analysis, impact analysis, decision of implementation, implementation, regression testing, acceptance, closure of the intervention and re-insertion.

The Institute of Electrical and Electronics Engineers (IEEE) has published a standard for software maintenance that again lays down definitions and describes a process with seven stages: problem identification, analysis, design, implementation, system test, acceptance test and delivery. Inputs, outputs, process, controls and metrics are briefly described for each stage (IEEE, 1993).

From the point of view of someone who wants to create a process, the main problem with these sources is the level of detail. They tend to model at a fairly high level how someone thinks maintenance should be done, not precisely how it really has been done in any particular setting. While a great deal of experience has gone into constructing these models, much of that experience has been abstracted away in the published work.

In designing and using GUMP, we have found that our biggest problems have been with messy details, and that the literature has often provided very little guidance. Four of the key issues and the rationale for GUMP's solution in each case are described in the following subsections.

Configuration management and code locking

Any maintenance process that does not 'lock' code to prevent inconsistent updates risks introducing chaos (Babich, 1986). But locking code may require delaying programmers working on one *tisk* while another completes its work. The GUMP process locks code for a relatively long period (often eight weeks or more) between cell 500 and cell 800 (Figure 2). We have found that our project planning has been almost totally conditioned by these long lock times. We have to group *Deficiencies* into tisks that are as independent as possible, and schedule work so that the implementation and testing phase of one *tisk* does not overlap with the those of a following related *tisk*. Essentially, our priorities are driven not by the needs of users of our software, but by our configuration management system!

We adopted long-term locking to guarantee that only thoroughly verified code can get into the baseline. Some organizations use a short term code locking strategy, in which programmers make changes, partially test them, and then check them in on a trial basis within a day or so.

Programmers working on one module must repeatedly get the latest version of other modules to build and test; there is a danger that one programmer's error can cause mysterious failures in another's work.

The trade-off between long-term and short-term locking will depend on the number of simultaneous changes that are being made, the typical size and overlaps of the different

changes, and the ability of the programmers to work with complicated code interactions. GUMP is used in an educational context and many of our student programmers are relatively inexperienced so we adopted long-term locking, but the cost has been high and we doubt that most commercial organizations should take this same decision.

Detailed design and domain knowledge in the maintenance phase

The initial version of GUMP included a detailed design task as part of implementation (cell 600) but we have not been able to define clearly what such a task should involve. While the literature on software design is extensive, there seems to be very little guidance on how to do design during maintenance.

Roughly speaking there are two objectives in preparing a design document. First, if several programmers will share coding, then the design document defines the interfaces they must all follow. Second, a design document makes design assumptions visible so that they can be inspected as part of SQA procedures, and may thus allow some kinds of errors to be caught.

Most of our maintenance changes have been quite small, ranging from 100 or so lines scattered through the code to enhancements involving about 1 500 lines of code. These are one-person tasks so the first objective mentioned is not relevant. And even for the enhancement tasks, we have found that documenting the detailed design seemed to add work without contributing much error detection value.

Over the last 18 months, our most serious design errors have come not from low-level mistakes, but rather from misunderstandings of the user's needs or invalid assumptions about the way the software works. These errors are unlikely to be detected by writing down a more detailed design. Essentially we have come up against the problem of 'thin spread of application domain knowledge' so often cited by large development organizations (Curtis, Krasner and Iscoe, 1988).

The strategy we are now adopting in GUMP is to try to use the few domain experts most productively. Our students have a wide range of experience, so the process will give the most knowledgeable more opportunities to support the others. In the latest version of GUMP the detailed design task has been taken out of cell 600. Instead, analysts are required to sketch out their proposed solution in the *Analysis Report* during cell 300. This sketch is then reviewed three times; informally by peers while writing the *Analysis Report*; more formally under SQA supervision in a final Analysis Report Inspection, and finally by the Change Control Board with management and client representation. We hope that by concentrating our experts in these three reviews we will be able to cut down on our most important source of error.

Quality assurance procedures

Quality Assurance techniques are intended to reduce the overall number of software errors and to detect them as early as possible. The main techniques are inspections and testing, but unfortunately both are quite expensive. On the one hand, such techniques should be used as early and as intensively as possible to control errors economically. On the other hand, each additional inspection or testing step significantly delays *tisk* completion. The literature provides little guidance on appropriate standards for inspections and testing in maintenance projects.

The current version of GUMP includes two inspection and two testing tasks. Inspections supervised by the SQA team are performed at the conclusion of Analysis (cell 300) and of Implementation (cell 600). Unit testing is performed by the software engineering team in cell 600 and independent integration and system testing is done by the V&V team in cell 700. Standards documents describe the goals and steps for each of these activities, though we do not yet have enough experience to develop useful inspection checklists.

These four checkpoints seem to be sufficient for the kinds of software change we have made so far; an additional inspection in cell 700 was dropped when it did not seem to contribute anything useful on the first few *tisks*. As we gain further experience we hope to develop more quantitative data to let us tune our quality assurance approach.

Allowing process flexibility

No process can be so thorough as to handle every contingency that may occur; some method is needed to allow flexibility while still preserving the benefits of a defined process. From our very first *tisk* we found some things that didn't go quite according to plan. A typical example was a change implementation that differed from the specification in the *Analysis Report* in small details because of minor difficulties encountered in coding the original concept.

Some industrial processes we have seen provide flexibility by allowing the basic process to be tailored before each project. Instead, the current GUMP process systematizes the idea of *variances* that may be introduced to cover such minor changes. A variance is typically stated in a one page memorandum. A software engineer requests a variance if some aspect of the process is inappropriate for a particular *tisk* or if some work product from a completed cell needs to be changed. However variances have to be publicly documented and approved by the Change Control Board to guarantee that any side effects of the change have been considered, that all the affected groups are informed, and that really dangerous problems are not being swept under the carpet.

TEACHING SOFTWARE ENGINEERING USING GUMP

With almost two years' experience, we now see a maintenance process as an almost ideal vehicle for teaching software process and software engineering management concepts. Most of our earlier student projects involved new software development but there was never an opportunity to perform each activity (requirements, design, coding, etc.) more than once. In a maintenance project, students cycle through the different process activities several times over the two semester period. This means that there is time for feedback and learning; if an analysis phase error is only detected in final testing then the group can decide how the process must be improved to prevent this same error next time round, and there is time to try out the modified process and see if it really works.

One major academic goal has been to teach students to emphasize software engineering discipline and not just shipping code. This goal seems to have been achieved. Students now put at least as much enthusiasm into improving the process as they do into the software itself. They concentrate on defining better ways of performing their jobs instead of on meeting an arbitrary deadline for delivering the product.

Another goal of the project course is to teach the appropriate use of software quality assurance techniques. In our earlier projects, just as often happens in industry, inspections

and testing were often slighted when deadlines approached. Now that we have a defined process to work with, students have a much better appreciation of what it takes, organizationally and technically, to produce a good product. On one occasion, the students evaluated the results of the inspections and independent testing and took the hard decision to 'roll back' a complete *tisk* since the quality was inferior. It would be hard to imagine any of our earlier project teams taking that decision.

One final experience has allowed students to see the concrete benefits of following a process. One enhancement *tisk* was completed during the summer of 1995 when only a few students were available, so the full process could not be applied. The new student team starting work in the fall of 1995 immediately identified the module produced by this *tisk* as the most buggy and worst coded in the entire system. The contrast between code produced with and without a process could not have been more dramatic!

AVAILABILITY OF GUMP DOCUMENTS

The complete GUMP software maintenance process consists of the documents listed in Table 2. These are available electronically by web browser at the following URL.

http://www.cs.uwf.edu/~wilde/gump/

If you have difficulty accessing them please send electronic mail to wilde@dcsuwf.dcsnod.uwf.edu. License is granted to copy, use and/or modify these documents as described in the 'readme' file stored with them.

Table 2. Documents of the generic UWF maintenance process (GUMP)

Document	Purpose
Process architecture	Provides universal and worldly process and ETX diagrams for each cell
Management	Incorporates management level items not found in other documents; single reference point for all documents
Software quality assurance	Describes when and how SQA will be performed during the software maintenance process; outlines all inspection procedures; provides inspection checklists
Software configuration management	Defines configuration management process activities for control of all software products, to include file and directory naming conventions, CCB procedures, and baseline configuration
Independent verification and validation	Describes standards, test cases to be developed, the IV & V procedures required, and the resulting reports
Unit test standards	Describes the standards for unit testing using the ATAC tool
Analysis report standard	Describes contents of the *Analysis Report*
Style guide	Describes the style to be used in GUMP documents

CONCLUSIONS

Processes are like faces; no two are alike. The GUMP process was developed to guide student teams maintaining a fairly small system in the context of a graduate program in software engineering. It still has many limitations. We would especially like to improve our collection and use of metrics, to improve both cost estimation and defect prevention. GUMP would not be optimal or perhaps even functional in other organizations or other circumstances but we hope that it may be, if not a model, at least a useful example for educators and for managers who are struggling with their own problems of process definition.

Acknowledgements

We would like to thank the reviewers for their very constructive suggestions for the improvement of this paper, and we would like to thank the following participants in the *Software Engineering Management* and *Software Engineering Project* courses, all of whom contributed their efforts and enthusiasm to developing and refining the process described in this paper: Susie Arnold, Robin Bridges, John Carlin, Neale Currie, Tom Devenny, Orlando Figueredo, Mark Holman, Clyde Hurst, Mike Jackowski, Thomas Marler, Diane Memory, Billy Miller, Suzanne Nagy, Gregg Palmer, Mark Reeves, Karen Sledge, John Suarez-Beard, Pete Tully, Kevin Bongiovanni, Scott Brown, Chuck Cooper, Kim Davenport, Tom Drake, Mark Gillott, Teri Hudson, Melanie Ivery, Leslie Love, Rob Matles, Brice Meyer, Gail Mitchell, Indara Outsama, Marcela Pohl, Stu Rodgers, Tammy Stone, Robert Thorn, John Williams.

References

Babich, W. (1986) *Software Configuration Management*, Addison-Wesley Publishing Company, Reading, Massachusetts, USA.

Capretz, M. and Munro, M. (1994) 'Software configuration management issues in the maintenance of existing systems', *Journal of Software Maintenance: Research and Practice*, **6**(1), 1–14.

Cherinka, R., Overstreet, C., Cadwell, A. and Ricci, J. (1994) 'Issues in software process automation—from a practical perspective', in *Proceedings of International Conference on Software Maintenance—1994*, Victoria, Canada, September 1994, IEEE Computer Society, pp. 109–118.

Curtis, B., Krasner, H. and Iscoe, N. (1988) 'A field study of the software design process for large systems', *Communications of the ACM*, **31**(11), 1268–1287.

Drew, D. (1992) 'Tailoring the Software Engineering Institute's (SEI) Capability Maturity Model (CMM) to a software sustaining engineering organization', in *Proceedings of Conference on Software Maintenance—1992*, Orlando, FL, November 1992, IEEE Computer Society, pp. 137–144.

Harjani, D.-R. and Queille, J.-P. (1992) 'A process model for the maintenance of large space systems software', in *Proceedings of Conference on Software Maintenance—1992*, Orlando, FL, November 1992, IEEE Computer Society, pp. 127–136.

Hinley, D. S. and Bennett, K. H. (1992) 'Developing a model to manage the software maintenance process', in *Proceedings of Conference on Software Maintenance–1992*, Orlando, FL, November 1992, IEEE Computer Society, pp. 174–182.

Horgan, J., London, S. and Lyu, M. (1994) 'Achieving softwate quality with testing coverage measures', *IEEE Computer*, **27**(9), 60–69.

Humphrey, W. (1989) *Managing the Software Process*, Addison-Wesley Publishing Company, Reading, Massachusetts, USA.

Humphrey, W. (1995) *A Discipline for Software Engineering*, Addison-Wesley Publishing Company, Reading, Massachusetts, USA.

IEEE (1993) *IEEE Standard for Software Maintenance*, IEEE Std 1219–1993, Institute of Electrical and Electronic Engineers, New York, NY.

Paulk, M., Curtis, B., Chrissis, M. B. and Weber, C. (1993) *Capability Maturity Model for Software,*

Version 1.1, CMU/SEI-93-TR-24, February 1993, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Wilde, N. and Scully, M. (1995) 'Software reconnaissance: mapping program features to code', *Journal of Software Maintenance: Research and Practice,* **7**(1), 49–62.

Authors' biographies:

Norman Wilde received his Ph.D. in mathematics and operations research from the Massachusetts Institute of Technology in 1971. He spent a number of years working in developing countries overseas: in academic positions, with the World Health Organization, and as an independent systems consultant.

Since 1986 he has been working with the Software Engineering Research Center on a series of projects in software maintenance and particularly in the area of dependency analysis and software reconnaissance and has worked closely with the research programs of several Center affiliates. He is the author of a series of research papers on software maintenance as well as a Software Engineering Institute curriculum module on dependency analysis.

Dr. Wilde is currently an Associate Professor with the Department of Computer Science at the University of West Florida in Pensacola, Florida. Email: wilde@dcsuwf.dcsnod.uwf.edu

Scott Brown received his undergraduate degree in computer engineering from the University of Minnesota-Duluth in February 1992, prior to receiving a commission in the United States Air Force. Lt. Brown received his Master of Science degree in computer science-software engineering from the University of West Florida in August 1995.

Lt. Brown is currently a graduate student in computer engineering at the Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio. His research interests include software process improvement, automated reasoning systems, and artificial intelligence methodologies.